ECE-593 SPRING 2024

FUNDAMENTALS OF PRE-SILICON VALIDATION

IMPLEMENTATION AND VERIFICATION OF ASYNCHRONOUS FIFO USING BOTH CLASS-BASED AND UVM METHODOLOGIES

VERIFICATION PLAN (Upto Milestone 4)

Section-1 Team-1

Kumar Durga Manohar Karna (kkarna@pdx.edu)
Mohammed Abbas Shaik (mohammee@pdx.edu)
Nivedita Boyina (nivedita@pdx.edu)
Nikhitha Vadnala (vadnala@pdx.edu)

Contents:

Objective of a Verification Plan:

The objective of a verification plan for the asynchronous FIFO(First-In-First-Out) design in System Verilog, is to ensure that the Design Under Test(DUT) meets its specifications and functions under specific conditions mentioned for its operation. The verification plan is designed to thoroughly validate the asynchronous data transfer, storage, and retrieval processes within the FIFO, ensuring it handles corner cases and error scenarios correctly. This includes extensive testing of asynchronous read and write operations, verifying data integrity, checking for overflow and underflow conditions, and ensuring synchronization between the write and read clocks, along with compliance with specified interface protocols. The plan includes simulations to assess both functional correctness and performance metrics, ensuring the asynchronous FIFO fits its intended role in a larger system. This involves defining the scope and coverage features, establishing verification goals, and specifying the test environment including tools, simulators, and testbenches. The plan outlines test strategies like directed and random testing, develops specific test cases and scenarios, and sets coverage metrics to measure progress, ultimately aiming for a thoroughly tested and reliable final product.

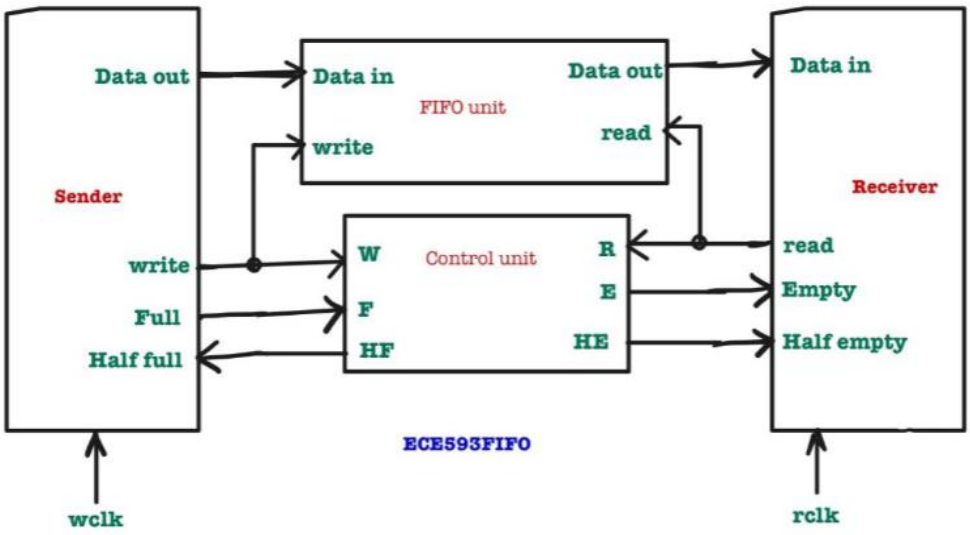
Introduction:

An Asynchronous FIFO (First-In-First-Out) is a specialized type of FIFO buffer that allows data to be read and written in different clock domains. Unlike a synchronous FIFO, where the read and write operations are controlled by the same clock signal, an Asynchronous FIFO will have separate read and write clock domains, enabling data transfer between two independently clocked systems.

The Asynchronous FIFO acts as a bridge between the sender and receiver modules, providing a reliable and efficient means of transferring data across different clock domains. It prevents data loss or data corruption during the transfer process, even when the write and read clocks are completely asynchronous to each other.

The design of an Asynchronous FIFO involves several challenges, such as metastability handling, gray code pointers, and synchronization techniques. Metastability can arise when data is transferred across asynchronous clock domains, which leads to undefined logic states. Proper metastability handling techniques, such as synchronizers, ensure reliable data transfer across clock domain boundaries. Grey cloud pointers with the property of changing only one bit at a time, which reduces the metastability. It also tracks the read and write positions in the memory. Additionally, the design incorporates synchronization techniques to properly coordinate the data transfer between the asynchronous clock domains.

Design Specifications:



Level Block Diagram of the Design System

To understand the FIFO design, we need to understand how the FIFO pointers work. The write pointer always points to the next word to be written; therefore, on reset, both pointers are set to zero, which also happens to be the next FIFO word location to be written. On a FIFO-write operation, the memory location that is pointed to by the write pointer is written, and then the write pointer is incremented to point to the next location to be written.

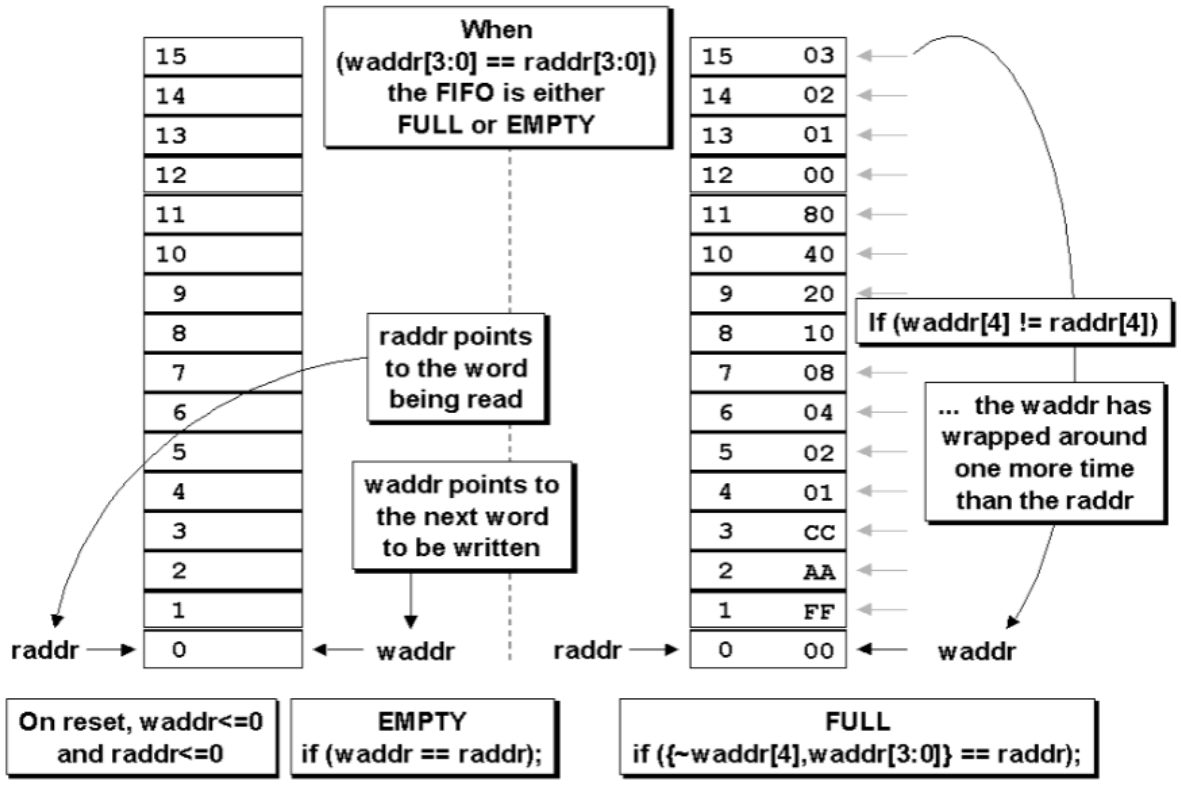
Similarly, the read pointer always points to the current FIFO word to be read. On reset, both pointers are reset to zero, the FIFO is empty, and the read pointer is pointing to invalid data (because the FIFO is empty, and the empty flag is asserted). As soon as the first data word is written to the FIFO, the write pointer increments, the empty flag is cleared, and the read pointer that is still addressing the contents of the first FIFO memory word, immediately drives that first valid word onto the FIFO data output port, to be read by the receiver logic. The fact that the read pointer is always pointing to the next FIFO word to be read means that the receiver logic does not have to use two clock periods to read the data word. If the receiver first had to increment the read pointer before reading FIFO data word, the receiver would clock once to output the data word from the FIFO and clock a second time to capture the data word into the receiver. That would be needlessly inefficient.

The FIFO is empty when the read and write pointers are both equal. This condition happens when both pointers are reset to zero during a reset operation, or when the read pointer catches up to the write pointer, having read the last word from the FIFO.

FIFO is full when the pointers are again equal, that is, when the write pointer has wrapped around and caught up to the read pointer. This is a problem. The FIFO is either empty or full when the pointers are equal, but which?

One design technique used to distinguish between full and empty is to add an extra bit to each pointer. When the write pointer increments past the final FIFO address, the write pointer will increment the unused MSB while setting the rest of the bits back to zero as shown in Figure 1 (the FIFO has wrapped and toggled the pointer MSB). The same is done with the read pointer. If the MSBs of the two pointers are different, it means that the write pointer has wrapped one more time than the read pointer. If the MSBs of the two pointers are the same, it means that both pointers have wrapped the same number of times.

Using n-bit pointers where (n-1) is the number of address bits required to access the entire FIFO memory buffer, the FIFO is empty when both pointers, including the MSBs are equal. And the FIFO is full when both pointers, except the MSBs are equal.



Tools Required:

Design simulation and verification done using QuestaSim

FIFO depth Calculations:

$f_A > f_B$ with idle cycles in both write and read.

Writing frequency = f_A = 120MHz.

Reading Frequency = f_B = 50MHz.

Burst Length = No. of data items to be transferred = 1024.

No. of idle cycles between two successive writes is = 4.

No. of idle cycles between two successive reads is = 2.

The no. of idle cycles between two successive writes is 4 clock cycle. It means that, after writing one data, module A is waiting for four clock cycle, to initiate the next write. So, it can be understood that for every five clock cycles, one data is written.

The no. of idle cycles between two successive reads is 2 clock cycles. It means that, after reading one data, module B is waiting for 2 clock cycles, to initiate the next read. So, it can be understood that for every three clock cycles, one data is read.

Time required to write one data item = $5 * 1/120$ MHz =41.67 ns = 42 ns

Time required to write all the data in the burst = 1024 * 42 ns. = 43008 ns.

Time required to read one data item = $3 * (1/50)$ MHz =60 ns.

So, for every 60 ns, module B is going to read one data in the burst.

So, in a period of 43008 ns, 1024 no. of data items can be written.

The no. of data items can be read in a period of 43008 ns = 43008 ns / 60 ns= 717

The remaining no. of bytes to be stored in the FIFO = 1024 – 717 = 307.

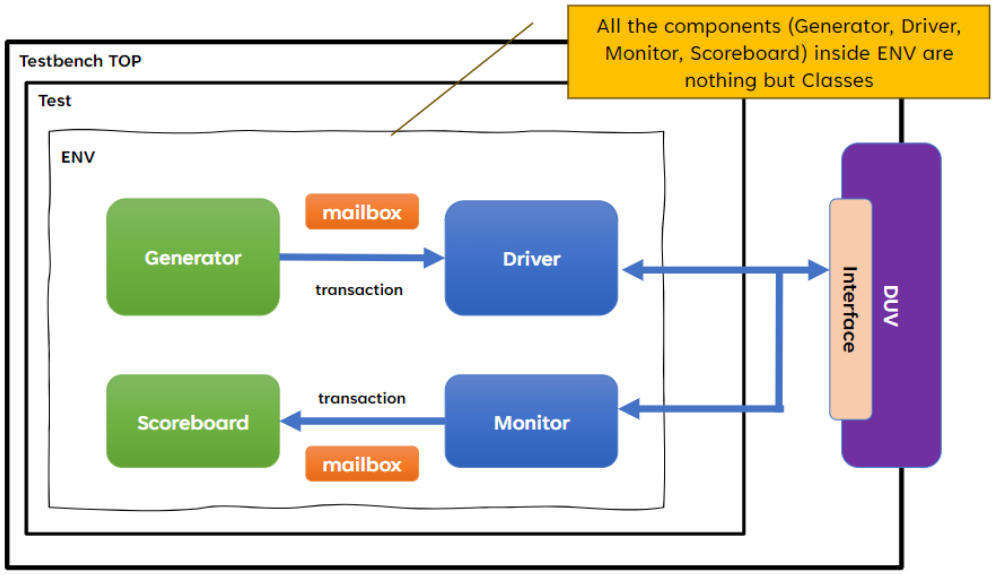
So, the FIFO which must be in this scenario must be capable of storing 307 data items.

So, the minimum depth of the FIFO should be **307**.

Duty cycle is around 50%.

Testbench Architecture in System Verilog :

For Class Based Verification



- **Generator:**

  Generator class is responsible for generating the stimulus by randomizing the transaction class and sending the randomized class to driver via mailbox. The transaction class handle is declared, randomized and an event is added to indicate the end of the generation of packets. We used “triggering” here to indicate end of the generation.

- **Driver:**

  A Driver will receive the stimulus from the generator and drive them to the DUT by assigning the transaction class values to interface signals. Adding a local variable to track the number of packets driven and increment the variable in the drive task.

- **Monitor:**

  A Monitor is used for sampling the interface signals and will convert the signal level activity to a transaction level. Once sampled, the transaction will be sent to the scoreboard using the mailbox. In the monitor class, we instantiate the virtual interface which allows to pass an argument to a method, then the extended class method handle will get assigned to the base class handle.

- **Interface:**

  An interface is a bundle of signals or nets through which a testbench communicates with a design. Allows the number of signals to be grouped together and represented as a single port. A virtual interface also allows you to pass an interface as an argument to a task, function, or method. This way, you can reuse the same code for different interfaces without changing the code.

- **Mailbox:**

  Mailbox is a communication mechanism in System Verilog that allows messages to be exchanged between process. Here, we have used 2 mailboxes. One is Generator to Driver and the other one is for Monitor to Scoreboard transactions. We use mailbox because it stores temporarily in system's memory object.

- **Transaction**:

  Fields required to generate the stimulus are declared in the transaction class. Here, we are giving input stimulus in randomized way, thereby making use of the "rand" keyword.

- **Scoreboard:**

  Scoreboard receives the sampled packet of the signals from the monitor via the mailbox and then compares each transaction with the expected result, an error will be reported if the comparison results in a mismatch.

- **Environment:**

  Environment class contains the Mailbox, Generator, Driver, Scoreboard, Monitor and Transaction classes. The communication between Generator and Driver are instantiated (mailboxes) and pass the interface handle. A task is added to call all the above methods.

- **Test:**

  The test is responsible for the creation of environment and also setting the number of transactions to be generated and initializing the stimulus driving process.

- **Testbench:**

  The testbench is the top module which connects the test module and the DUT module by making use of the interface. A clock is also generated in this module itself.

Coverage Details:

The coverage report generated for the top module provides a comprehensive analysis of the functional and code coverage achieved during the verification process. This report evaluates how thoroughly the testbench exercised the FIFO's functionality, focusing on critical aspects such as data transfers, clock domain synchronization, and handling of reset conditions.

Goal for Each Coverage Metric:

- **Statement Coverage**: Increase coverage to at least 90% to ensure that the majority of code lines are executed during simulation, identifying potential dead code.
- **Branch Coverage**: Aim for at least 80% coverage to ensure that critical decision points in the code are thoroughly tested, providing confidence in control flow.
- **Expression Coverage:** Target 95% coverage to ensure that most expressions are evaluated in both true and false conditions, verifying logical paths.
- **Toggle Coverage**: Maintain the goal of at least 90% coverage to ensure that the majority of signals in the design are exercised, capturing signal activity.
- **Data Transfer Coverage**: Ensure that all the data input and output scenarios, including various data sizes and types, are tested to verify accurate data transfer between the FIFO and external modules.
- **Flag Coverage**: Validate the correct behavior of flag signals such as **wFull** (write full) and **rEmpty** (read empty) under different load conditions and operations.
- **Clock Domain Crossing Coverage:** Verify proper synchronization between write and read clock domains, ensuring data integrity during asynchronous operations. Achieve 100% coverage for clock domain crossing scenarios, including different clock frequencies and phase misalignments.

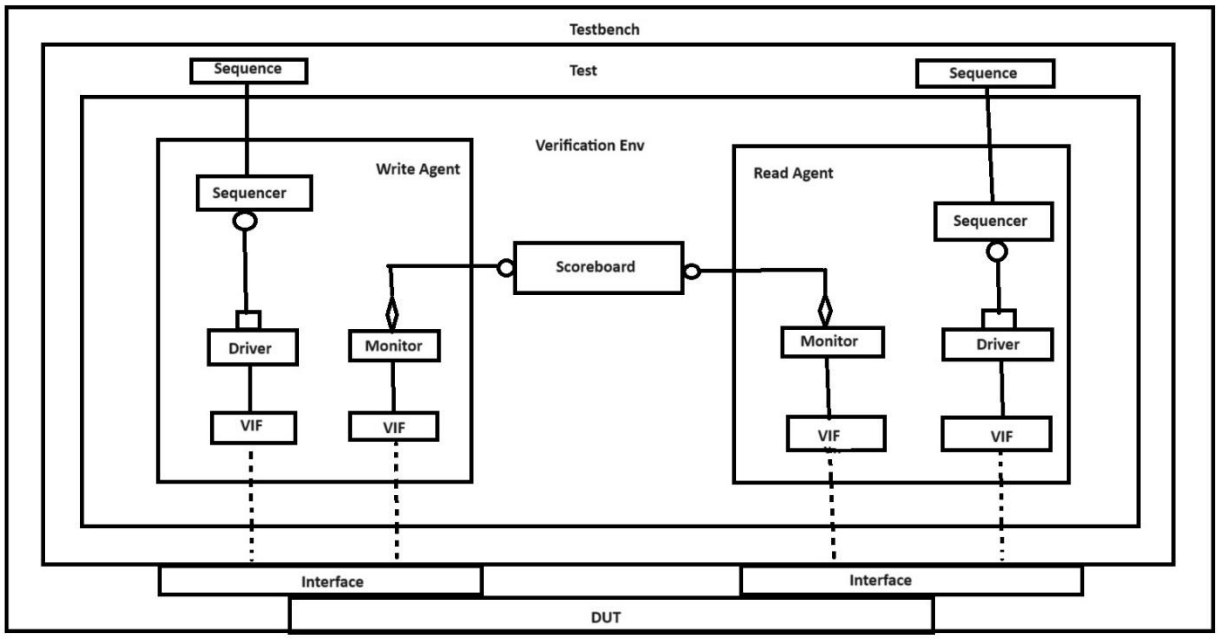
Coverpoints and bins:

- Code coverage Covergroup one for each module o Individual coverpoints for all lines of code . Bins that will ensure all lines are covered during simulation.
- Branch coverage Covergroup Coverpoint that targets any decision branching statements. Bins will exercise each statement high and low.
- Covergroup for Top read and write operations. Coverpoint for different data transfer conditions. Bins that ensure all possible data bursts are accounted for.
- Toggle Covergroup coverpoints that track signal toggle. Bins that ensure transitions from 0 to 1 and 1 to 0.

Testbench Architecture in Universal Verification Methodology (UVM):

The UVM (Universal Verification Methodology) testbench architecture consists of several components that work together to generate stimuli, apply them to the DUT (Design Under Test), and check the DUT's responses. The architecture is designed to be modular and reusable, allowing for efficient and thorough verification.

For UVM based verification



UVM Components

- **Test:**

  It is the highest level of the testbench which is responsible for instantiating and for configuring the environment. It controls the overall test flow. It initiates the stimulus by starting the sequence.

- **Environment:**

  The environment manages the configuration and coordination of the testbench components. It contains multiple agents for different interfaces, a scoreboard common for these agents and a coverage collector for functional coverage. It is a container component for grouping higher level components like agent's and scoreboard.

- **Agent:**

  UVM agent groups the UVM components specific to an interface. It encapsulates Sequencer, Driver and Monitor into a single entity by connecting via TLM interfaces. Agents are of 2 types. Active and Passive

- **Sequencer:**

  A sequencer generates sequence of transactions. It controls the flow of the transaction data to the driver for execution. Unless you have additional ports to be included, you should directly instantiate it as a "uvm_sequencer" parameterized to the required data item.

- **Driver:**

  UVM driver is an active entity that drives the generated stimuli onto the DUT's interface. It also converts the high-level transactions into pin level signals. Also, all the driver classes should be extended from "uvm_driver", either directly or indirectly.

- **Monitor:**

  Monitor will observe the signals on the DUT's interface and convert into packet level signals which is sent to components like scoreboard and coverage collector. It may have knobs to enable/disable basic protocol checking and coverage collection.

- **Scoreboard:**

  UVM scoreboard is a verification component that contains checkers and verifies the functionality of a design. It will compare the expected values with the actual values to determine correctness. It typically implements a data integrity check.

- **Coverage Collector:**

  The UVM test bench has a common coverage collector for collecting functional coverage data. It helps in measuring the completeness of the verification process. It receives the transactions from the Monitor and updates the functional coverage metrics to ensure all scenarios have been tested.

- **Transaction:**

  The transaction here represents the data packet or operation being sent to/received from the DUT. The basic unit of communication between sequencer, driver and monitor.

- **Interface:**

  The interface defines the connection between the testbench and the DUT. It contains the signals and the physical implementation of the communication protocols. If we define and use interfaces properly, we can enhance the modularity and reusability of the verification environment.

Test Plan:

Test Name	Description	Expected Results
Basic Read and Write operations	To check Read and write into array at module level	Data written to the FIFO should be read back in the same order
Overflow condition	To test the FIFO's behavior when the write pointer reaches the maximum depth	The "wfull" flag should assert when the FIFO is full.
Underflow condition	To test the FIFO's behavior when the read pointer exceeds the available data.	The "rempty" flag should assert when the FIFO is empty.
Reset Behavior	To verify the FIFO's response to the reset signals (wrst and rrst)	All pointers should reset to initial values
Write and Read with Idle cycles	To validate the FIFO operation with idle cycles between successive reads and writes	Data should be correctly read and written despite idle cycles
Coverage metrics	To check that all coverpoints and cross-coverage points defined in the coverage model are hit or not.	Coverage report should show complete coverage, including all scenarios
Incremental Signal Functionality	To test functionality of the Write increment and read increment	Write and read pointers should update correctly reflecting data writes and reads.
Half full and Half empty	Verify behavior of FIFO when it is Half full and Half empty	Confirm that the FIFO correctly signals its full and empty states when it reaches the halfway mark.

Created different classes in the testbench environment and implemented the class-based verification for the DUT. Generating various randomized stimuli in the generator and verifying it for number of transactions.

Contribution:

Kumar Durga Manohar Karna - Led the implementation of the **top** module, including parameterization and instantiation of internal components. Developed the clock and reset logic for the FIFO module, ensuring proper initialization and synchronization. Integrated the FIFO module into the testbench environment using UVM components.

Mohammed Abbas Shaik - Conducted functional verification of the FIFO module, designing and executing test cases to validate data transfer, flag signaling, and reset behavior. Analyzed code and functional coverage reports to assess the completeness of verification efforts and identify coverage gaps for UVM components too. Identified and resolved issues encountered during simulation, ensuring the correctness and reliability of the FIFO design.

Nivedita Boyina - Implemented coverage-driven verification strategies to ensure comprehensive testing of the FIFO module, including statement, branch, and expression coverage. Collaborated with other team members to coordinate verification efforts, share insights, and address challenges encountered during the project. Documented verification plans, UVM architecture, test cases, and results, and prepared detailed reports highlighting verification progress and outcomes.

Nikhitha Vadnala - Have gone through the design specifications and created a class for scoreboard which includes the read and write operations with half full, half empty, full and empty conditions and displays the output. Have created an interface for both class based and UVM testbench and have implemented clocking for driver and monitor. Led the development of functional coverage models for our asynchronous FIFO module, meticulously capturing critical features and scenarios to facilitate thorough testing.

Resources:

1.http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf

2.http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf

3.https://hardwaregeeksblog.files.wordpress.com/2016/12/fifodepthcalculationmadeeasy2.pdf

4.https://ieeexplore.ieee.org/abstract/document/7237325

GitHub link:

https://github.com/BNiVeDiTa29/ECE593s24_team_01_Async_FIFO